Extending Swarm Communication to Unify Choreography and Long-lived Processes

Lenuta Alboaie *adria@info.uaic.ro*
Faculty of Computer Science of the University "Al. I. Cuza"
Iasi, Romania

Sinica Alboaie *abss@axiologic.ro*
S.C. Axiologic SaaS
Iasi, Romania

Tudor Barbu *tudor.barbu@mail.academiaromana-is.ro*
Institute of Computer Science of the Romanian Academy
Iasi, Romania

Abstract

The usual way of doing message passing is to have relatively intelligent processes, objects or actors sending and receiving dumb messages. Swarm communication presents the inverted approach of thinking to messages as relatively smart beings, visiting relatively non intelligent places, to produce better code and a new path for integrating complex applications. Swarm communication can be used as software architecture for an Enterprise Service Bus. This paper extends the swarm communication to support long-lived processes, providing an alternative approach to the standard techniques for executable Business Process Management. The ubiquity of connected devices is a fertile ground for new approaches to application development and systems integration based on complex business processes.
Keywords: Long-lived Processes, Communication Pattern, Message Passing, Asynchronous Communication.

1. Introduction

Combining services to create higher level, cross-organizational business processes, requires pragmatic approaches to model the interactions. Existing standards for service oriented architecture [5] and formalized workflows [19], are used by big companies on big projects but are quite complex and expensive for smaller companies because of the number of required skills. Therefore, many applications are still created with a monolithic architecture and by hard coding business processes in imperative code. Specific languages to describe long-lived processes, orchestration technologies are not in the toolbox of the typical programmers. The growing popularity of micro-services [10], a style of software architecture that involves systems based on a set of very granular and independent collaborating services, require new approaches and pragmatic languages for integration, coordination and orchestration. Ideally, one new language that could solve all the aspects of integration, coordination and orchestration without involving a big learning curve for existing programmers could have huge economic benefits and a large adoption. We already proposed swarm communication as a new type of distributed programming technique in [1]. Swarm communication is an asynchronous message programming pattern. Also, our proposal includes an architecture created for software integration, especially an architecture for doing choreography in an ESB (Enterprise Service Bus) [4]. In our swarm model, our vision over orchestration and choreography match with those presented in [18] "Orchestration differs from choreography in that it describes a process flow between services, controlled by a single party. More collaborative, choreography tracks the sequence of messages involving multiple parties,

where no one party truly "owns" the conversation." Commonly, BPM (Business Process Management) technologies [6] are deployed in ESBs as a service connected to the BUS and they perform orchestration. In this article we show how we can extend the swarm communication to model also long-lived business processes [18], the kind of process that are usually implemented using BPM technologies like BPEL. We use long-lived processed term as expressing the most important aspect of workflow concept used in software systems. By providing a usable high level abstraction and possibility to express long-lived processes in a choreography language, we can replace the use of other orchestration components, significantly simplifying the programming efforts for SOA implementations.

In the next section we will make an overview of swarm concepts and constructs. Also, we will describe our Domain Specific Language [7] for swarms and we provide an implementation sample. In Section 2, we provide the most relevant technical details regarding swarms that help us to argue in Section 3 a new method for modeling long and short-lived processes using swarm communication. This paper finalizes with a section of conclusions and a list of references

2. Swarm Concepts and Constructs

2.1 Overview and the Context

Swarm communication brings a new perspective in the same area where we find well known models such as: actor model [10, 11], Message-Oriented Middleware (MOM) [3] or communications patterns found in systems like ZeroMQ [12] and Enterprise Integration Patterns [13]. According to [21] MOM is a middleware infrastructure that provides messaging capabilities. Our approach, as MOM, reduces the application developers' effort, but is different in terms of communication mechanism.

Programming with asynchronous messages is performed intuitively by using "send" and "receive" primitives or by setting the behavior of the actor on the next message received in the actor model and derivatives. Almost all practical approaches as programming with call-backs, OOP, Scala [16], Erlang [14], observer patterns [9] can be seen as derivatives of the formal actor model. We have shown in [3] that programming with asynchronous messaging can be made by describing chains of message transmissions with behaviors associated for each message transmission, instead of associating behavior within the receiving actor. In our approach an entity that receives messages (actors in usual terminology) still exists but an actor behaves like an idempotent service and not as an active communication participant that changes its state after receiving a message. The simple act of describing the behavior within message and not within actor description can have beneficial practical implications in code maintenance. The behavior is not scattered along multiple actors but in a single place. This way is easier to describe a program that can run in many distributed nodes. While theoretical research on mobile code [2] and on systems for asynchronous calculus exists by many years, our proposal is a practical approach that can be appealing for programmers used to program in mainstream languages Java, C#, Java Script, et al. and that will not easily switch to research programming languages (actor inspired languages, pi calculus et al.).

With swarms, the messages themselves have a long time identity during multiple communication events and an existence during complex communication processes. Messages are changing their state after each communication step.

In all other approaches a message does not have identity or associated behavior. Furthermore identity, state changes or behaviors are associated only with the message receivers (actors). This is the main conceptual difference of our proposal compared with the actor model.

In our vision, a swarm is a set of related messages with some basic intelligence and is based on an intuitive point of view: computer processes communicating by asynchronous messages are more like "dumb trees/flowers" visited by "smart swarms of bees" than "smart nodes" communicating with "dumb messages". In our approach, as we can see in the next

section, we make a rigorous identification of communication phases, and this communication became a choreography which ensures an overall effect on system and not just a simple communication through messages.

2.2 Concepts and Constructs

Swarm communication intends to keep the benefits of asynchronicity such as scalability, high availability, highly parallel computing and loose coupling, while decreasing the complexity and associated costs. A communication use-case can be viewed as a set of complementary messages which are sent in the distributed system to accomplish a "goal". This set of complementary messages, represented by a swarm, represents the flow of communication between some nodes in a distributed system when a use case happens.

Below we describe the main concepts and constructs that are involved in swarm communication. Any software entity that can be seen as receiving or sending swarm messages represents a *node*. A node is talking to another node by sending a swarm. When we talk by a node sending a swarm we mean that it sends a message that is part of the set of swarm messages. Nodes are materialized by client applications, adapters for existing server-side applications or swarm service providers. In our system we distinguish two types of nodes:

- *Adapters* are server side nodes that provide some services or APIs for an existing application or systems (e.g. ERP, CRM, etc.). Swarm service providers are considered also adapters because usually they use some other low level services (such as NoSql databases, message queues, et al.)
- *Clients*: all applications that communicate by sending swarm messages, excepting adapters, will be called *swarm clients* and are logically connected to an adapter by communication protocols created over TCP sockets or other protocols. In our current experiments we have three possible platforms available: Action Script (Flex) clients, node.js clients and web sockets clients from browser and REST clients. [20]

In order to describe swarms in our implementation we proposed Swarm DSL as a Domain Specific Language. For syntax description we used Backus-Naur Form notation. Swarm DSL is an internal DSL [10] so all Java Script syntactic and semantic rules should be considered. By using an internal DSL we can benefit from existing tools for debugging, IDEs (Integrated Development Environment) and programming expertise, therefore we reduce adoption risks for this new technology.
SwarmDSL language is presented below:

```
swarmDescription::="{" meta",”list”}” ","list"}"
list             ::= declaration{","declaration}
declaration      ::= vars | function | phase | ctor
meta ::= "meta" ":" "{"property {",”property} "}"
vars ::= "vars" ":" "{"property {",”property} "}"
function ::= identifier ":" "
             function(" varArgs ")" "{" code "}"
phase ::= identifier ":" "{"node","phaseCode "}"
ctor  ::= function
varArgs::= ""| identifier { "," identifier }
phaseCode ::= "function()" "{" code "}"
property ::= identifier ":" string | identifier ":" identifier
node     ::= "node" ":"  nodeName
nodeName ::= '"' "#" innerNodeName '"' | '"' wellKnownNode '"'
           | '"' "@" groupName '"'
```

Therefore, a swarm description can contain four construction types:

-*vars* section in swarm description is a way to document and initialize the variables used by the swarm messages

-*swarm phases* represents the swarm progress towards accomplishing a goal. Such phases contain applications code that change the internal variables of the swarm or the state of the adapters in which the swarm got executed. It is important to notice that the phase's code is

not part of the adapter, but code of the swarm itself, even if it is executed in the context of the adapter. Except that having a name (*phase name*) and a node hint (an indication of the node where the swarm should be executed), a phase is the behaviour (code) that should happen (execute) when a message is received by a node (adapter). A single swarm can execute code concurrently in multiple nodes and therefore multiple phases can be concurrently "alive". Swarm variables are not automatically shared between those concurrent executions, remaining to programmers exercise to implement communication between swarms.

- *meta variables* are special variables that are reflecting the current execution of the swarm and can be handled by programmers to influence or control the execution of the swarm in various ways.

- *functions and Ctors: Ctor functions* or *ctors* are functions that are called when a swarm is started (similar to constructors in OOP languages). A *ctor function* will initialize the swarm variables and will start swarming in one or multiple phases. Normal functions are utility functions made available to swarms' code for a greater modularity and code reuse between phases: *swarm()*, *startSwarm(), startRemoteSwarm(),home(),broadcast() et al.* In the context of this article we will do an overview only of *swarm()* primitive. We can view the swarm call as same as with Unix fork [17] for the current swarm abstract process because variables are duplicated. Therefore when a new child swarm is born in the context in which a swarm is a collective entity and not an individual entity, we can consider this new child swarm as part of the current swarm. The state of current execution of the swarm, at the moment of the call, is serialized and a message is sent to another node requesting execution of a specific phase to take place there. The node, where this new child swarm will continue, is decided by looking at the phase declaration. The parent swarm can continue to call the swarm primitive as many times as necessary or eventually it will end the execution.

In our DSL each node has a unique identifier in the system that we usually refer as *node name*. As described in BNF grammar, *nodeName* can have three possible values. *Inner* nodes are using resources from a normal node but they do not have any visibility outside of that node. Sending a swarm to an *inner* node does not produce network traffic. Sending a swarm to an inner node is exactly like sending the swarm to the current node, this feature can be used during development for comparing execution times between remote and local communication or when one plans what services (adapters) are required.

wellKnownNode can take as value a well-known node name. Such nodes represent the infrastructure nodes for our application (e.g. *Core* - the node that is the source authority for all swarm descriptions, *Logger* - the node where are stored logging information from all nodes, *SessionManager* that is responsible for managing nodes that keep information about sessions).

Well known nodes can be problematic for scalability (on high load could cause bottlenecks) and can be used for a while till the load increases and should be replaced by groups of nodes. Therefore to enable load balancing, replication and high availability we have introduced the concept of *groups*. Each adapter can join a number of groups by using a *join()* primitive. By joining a group a node becomes accessible to nodes that know the group name but are not aware of its name.

In our grammar we assume that *identifier* and *code* are valid JavaScript identifiers and respectively valid JavaScript source code that can appear in valid Java Script functions.

2.3 Swarm Example

In this section we present a "*swarmified*" Hello World example using Swarm DSL:

```
helloWorldSwarm={
       vars : { message : "Hello World" },
       meta:{ debug : false },
```

```
        start : function(){ this.swarm("concat");}
        concat   : { //begin phase
            node : "Core",
            code : function (){
              this.message = this.message + "The swarming has begun!";
              this.swarm("print"); }},
        print : { //print phase
            node :"Logger",
            code : function (){
            console.log(this.message);},}
}
```

In this example the swarm execution contains two phases: *concat* and *print*. The phase's execution is performed in two different nodes *Core* and *Logger*. These nodes are different processes, possibly located on different machines, depending on system configuration. A swarm starts through a *ctor* execution and this can be done on any node of the system. For our example, the constructor is called *start* and his role is to lead the swarm in *concat* phase using the *swarm()* primitive. According to our discussion from section 2.2, as we can notice in our example, each phase declaration has two components: *node* and *code.* In *node* we specify the location where the phase will be executed. To exemplify concurrency we should understand that even if a new phase is running, the current phase could still exist for a while. Of course, the variables in these two concurrent phases are not shared (are just copies) but this mechanism usually serve us well without any risks of interferences or performance penalties. In this article we will not go into further details of implementation, but we want to exemplify the business use cases for integration and long-lived business processes.

3. Extending Swarm Choreography with Long-lived Processes

We have already explained how swarm communication could be useful to create business process for software integration use cases. Before this paper, swarm communication were envisioned as short-lived processes that visit various nodes (adaptors) and the use cases of long-lived processes were not swarms' concern. The obvious method in obtaining long-lived processes with swarms is to save the state of a swarm message until some specific event or change happens. When such event appears, the swarm is awaked in a form of a message that gets sent and the desired phase will be executed. To preserve scalability and to have control on error management, we decide that swarm persistence should not be available to all nodes, but only in dedicated nodes that perform only this thing. We consider for this proposal an adapter named SwarmBPM is in charge with persisting in a waiting list all the swarm messages used for representing steps in long lived processes.

We are aware that BPM technologies can have three different directions: human-centric BPM, integration-centric BPM and document-centric BPM and we present in this article results that can cover the software architecture aspects for creating systems for these directions. If various automated business events or human interaction with business processes are reflected in software, we can safely assume that they change the state of some identifiable resources accessible for SwarmBPM adapter. Such resources will be formalized in our proposal as *global objects*. A global object is identified by a *global type* (a string that can be a schema URI or other similar identifier) and a unique id value. A long-lived process is usually connected to one or more entities (global objects) used in the integrated system (like a document, a command from a customer and any other entities relevant for business processes).

SwarmBPM implements three functions available to swarms: *registerSwarm (swarmSerialisation)* persist the swarm message that represents a step in a long-lived process until a relevant event occurs, *globalObjectChanged(objectType, objectId, serialisedValue)* inform the engine that a global object changed its state (some of its properties changed), *wakeUp(swarm)* launch a sleeping swarm and remove from its waiting list.

An infrastructure swarm description named *GlobalObjectChange.js* is used to inform the SwarmBPM on changes in the state of the global objects by calling *globalObjectChanged()* when the state of global objects changes. The components that are launching

GlobalObjectChange.js swarms in execution are usually the persistence layer part of various integrated sub-systems (using Aspect Oriented Programming techniques, database change monitoring or other techniques that are usually used in BPM implementations to create relevant higher level events). Until now, SwarmBPM have a list of sleeping swarms and is notified about changes in global objects. In order to map global objects changes to sleeping processes we extend swarm descriptions with an additional construction: *global* sections.
global sections is an object having as properties objects with two members: *objectTypeName*, *objectIdField*. The *objectTypeName* take as value a string specifying a *global type* and the *objectIdField* specifies the field name in the global object that is used as the *id* of the global object. Those properties are named with the same names as a swarm variable that is available for phase's code.

In order to make the difference between phases used in short-lived processes and those belonging to long-lived processes we extend phase declaration with an optional function declaration named *watch*. We have the convention that a phase that have a watch is sent first to the SwarmBPM and only SwarmBPM can decide to proceed further with the swarm phase execution. The *watch* function is called automatically by the SwarmBPM engine at any state change in an global object from the list of the global objects declared in the global section of the swarm. If the watch function is calling *wakeUp(this)* function then the swarm is awaked and delivered to other adapter. For complex cases requiring coordination between multiple global objects, our extension provide support for implementing complex business rules and complex workflows at the level of integration between services or application modules without other orchestration mechanisms in place.

In the next example, we assume that in the swarm variables we have a *document* variable representing a document instance belonging to a Document Management application and a *crmRecord* belonging to a CRM (Customer Relationship Management) application. This swarm description presents a long-lived process that automatically updates secrecy behavior in a CRM application when the secrecy field in the document permissions is modified in the Document Management.

```
secrecyWatcher = {
    vars: {document:null, crmRecord:null},
    global:{
        document:{
            objectTypeName: 'Document Management/Document',
            objectIdField: 'documentId'},
        crmRecord :{
            objectTypeName: 'CRM/RelatedDocument',
            objectIdField: 'relatedDocumentId' }},
    createCRMSecrecyProcess:function(document, crmRecord){
       //launched once for every document
        this.document       = document;
        this.crmRecord      = crmRecord;
        swarm("establishDocumentSecrecyInCRM");},
    establishDocumentSecrecyInCRM:{
        node:'CRM',
        watch: function () {
          if (!this.savedDocument ||
          this.document.isSecret != this.savedDocument.isSecret) {
               //the document has a changed status about secrecy
               wakeUp(this); //wake up and execute this phase }},
    code: function () {
   //update the state using the API provided by the CRM adapter
   crmApi.makeUpdateRecordSecrecy(this.crmRecord,
               this.document.isSecret);
   //save the document for future comparations
   this.savedDocument = this.document;
   this.savedDocument.isSecret = this.document.isSecret;
   // keep the process alive, watch for future changes in secrecy
   swarm("establishDocumentSecrecyInCRM"); }}}
```

The previous example can be also interpreted as a business rule, implemented as a long-lived process at the level of the ESB. When the business analysts are more inclined to model business processes as rules and not as long-lived processes, the swarm descriptions will have a circular structure to keep the rule active as in above example. Complex cases could be eventually accompanied by another level of modeling the logic and dependencies in global objects belonging directly to SwarmBPM using the data structures and imperative or functional code.

4. Comparison with Existing Approaches

There is no shortage of standard techniques and programming models for integration (Enterprise Integration Patterns), orchestration (WS-BPEL) and various workflow programming models (as executable Business Process Management), choreography (WS-CDL). The swarm communications and the extension proposed in this article, offers a unified programming model that is capable to tackle all these aspects. These unifications provide a new programming technique with low learning curve for creating systems that have the benefits of being scalable in performance, flexible in adding or removing components, having a loosely coupled design. Mastering all the existing different techniques for integration, orchestration, choreography and composing them in a distributed system takes years of study. With swarm communication we envision that we can get the same benefits after only a few weeks of learning. With less than ten new concepts that are based on an intuitive abstraction over asynchronous messages, the learning curve for swarm communication is an order of magnitude lower. The number of concepts of the stand technologies compared with the number of concepts proposed by swarm communication represents a good indicator for the learning curve. Mastering EIP concepts alone require understanding of at least 56 abstract concepts [13].

Swarm communications provide the additional benefit of reducing application development efforts, because provides a natural environment for creating and composing microservices [8]. Microservice architectural style is an approach to develop applications as a suite of small services, each running in its own process. Microservices can be created independently by different teams with different technologies and can be tested in isolation. Redundancy and recovery from failures can be more easily implemented because the application is broken in small, independent processes. Heterogeneous systems integration and implementations of business processes and business rules can be performed by swarm communication between processes (called adapters) that are holding microservices.

5. Conclusions

This paper presents swarm communication as an alternative representation of asynchronous message communication in a distributed system. We emphasize that swarms are not only a set of related messages but a powerful mechanism to describe short and long-lived business processes. A swarm can be regarded as a choreography mechanism for services or adaptors located in a distributed system. Apart of performance and scalability benefits obtained by having asynchronous messages at runtime, a swarm description is an abstraction that will provide a single place where complex processes get described. Swarm based systems are constructed as two definite layers: infrastructure source code found in swarm adapters and code implementing rather ephemeral integration and business process requirements. Thus we will maintain the original architectural clarity even in the presence of numerous business changes in requirements that is an expected reality in the applications lifetime. The source code in swarm adapters is an instance of following Open/Closed principle [15] which is an important predictor for reducing costs in development of the software systems.

As in any distributed system, failures can be a problem for swarm communication too. Redundancy and fail recovery support in our implementation consists now in an experimental mechanism that is based on the idea that the node sending a child swarm waits for a confirmation. This mechanism will be extended and future studies of the swarm systems in

the presence of failures will be conducted. In this paper we are extending the swarm communication to support long-lived processes and we opened also an opportunity to study distributed transactions for swarm based systems.

References

1. Alboaie, L., Alboaie, S., Panu A.: Swarm Communication - a Messaging Pattern proposal for Dynamic Scalability in Cloud. In: Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications (HPCC 2013), Zhangjiajie, China, (2013)
2. Carzaniga, A., Gian Pietro P., Vigna, G.: Designing distributed applications with mobile code paradigms. In: Proceedings of the 19th international conference on Software engineering, ACM, (1997)
3. Curry, E., Chambers, D., Lyons, G.: Extending message-oriented middleware using interception. In: Proceedings of the 26th International Conference on Software Engineering - W18L Workshop "International Workshop on Distributed Event-based Systems (DEBS 2004)", pp. 32 – 37, (2004)
4. Chappell, D.A.: Enterprise Service Bus: Theory in Practice, Publisher: O'Reilly Media, (2009)
5. Erl, T.: SOA Design Patterns, Publisher Prentice Hall, ISBN: 013613516, 9780136135166, (2009)
6. Fiammante Marc, Dynamic SOA and BPM: Best Practices for Business Process Management and SOA Agility,Publisher:IBM Press, ISBN-10: 0-13-713084-8 (2009)
7. Fowler Martin, Domain-Specific Languages, Publisher Pearson Education, ISBN: 0131392808, 9780131392809, (2010)
8. Fowler Martin, http://martinfowler.com/articles/microservices.html (2014)
9. Gert, F., Meijers, M., Pieter Van Winsen: Tool support for object-oriented patterns,Object-Oriented Programming. Springer Berlin Heidelberg, 472-495, (1997).
10. Hewitt, C.: Actor model of computation: Scalable robust information systems, preprint arXiv:1008.1459, (2010)
11. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. Proc. of 3rd Conference on Artificial Intelligence, pp. 235-245, (1973)
12. Hintjens, P.: ZeroMQ: Messaging for Many Applications, Oreilly and Associate Series, Publisher O'Reilly Media, Inc., ISBN: 1449334067, 9781449334062, (2013)
13. Hohpe G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Publisher Addison-Wesley, (2012)
14. Kessin, Z.: Building Web Applications with Erlang: Working with REST and Web Sockets on Yaws, Publisher O'Reilly, ISBN 1449309968, 9781449309961, (2012)
15. Meyer, B.: Object-oriented software construction, Vol. 2, New York: Prentice Hall, pp. 331-410, (1988)
16. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, Publisher Artima Inc, ISBN: 0981531601, 9780981531601, (2008)
17. Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, http://pubs.opengroup.org/onlinepubs/009695399/functions/fork.html (2004)
18. Peltz Chris: Web Services Orchestration and Choreography, Journal Computer, Volume 36 Issue 10, pp. 46-52, IEEE Computer Society Press Los Alamitos, CA, USA (2003)
19. Sanjiva, W., Curbera, F., Leymann, F., Storey, T., Ferguson, D.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More, Publisher: Prentice Hall; ISBN-10: 0131488740, (2005)
20. SwarmESB, https://github.com/salboaie/SwarmESB, (2013)
21. Qusay, M. (Editor): Middleware for Communications, pub. John Wiley & Sons, ISBN: 978-0-470-86206-3 (2004)